move\_base源码分析CPP：

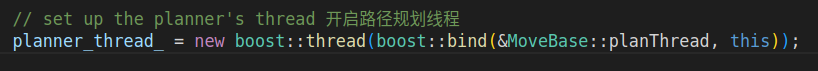
其实主要包括三个函数，第一个函数是move\_base( buffer )是MoveBase的构造函数，这个函数也是导航规划的入口函数。第二个函数和第三个函数都在move\_base( buffer )函数内，初始化并执行。第二个函数是executeCb()，是主线程，通过创建以下服务器来管理：



通过以下执行：



这个线程会进行局部路径规划计算控制指令，并发布话题。第三个函数是planThread()函数，是一个子线程，用于全局路径规划，获得全局路径。



1. movebase()

MoveBase::MoveBase(tf2\_ros::Buffer &tf) : tf\_(tf),//初始化一些参数

as\_(NULL),

planner\_costmap\_ros\_(NULL), controller\_costmap\_ros\_(NULL),

bgp\_loader\_("nav\_core", "nav\_core::BaseGlobalPlanner"),

blp\_loader\_("nav\_core", "nav\_core::BaseLocalPlanner"),

recovery\_loader\_("nav\_core", "nav\_core::RecoveryBehavior"),

planner\_plan\_(NULL), latest\_plan\_(NULL), controller\_plan\_(NULL),

runPlanner\_(false), setup\_(false), p\_freq\_change\_(false), c\_freq\_change\_(false), new\_global\_plan\_(false)

{

as\_ = new MoveBaseActionServer(ros::NodeHandle(), "move\_base", boost::bind(&MoveBase::executeCb, this, \_1), false);

// as\_维护movebase的actionServer状态机，并且新建了一个executeCb线程

ros::NodeHandle private\_nh("~"); // private\_nh的命名空间应该是 move\_base/node\_name(

ros::NodeHandle nh; // nh在的命名空间就是 /move\_base

recovery\_trigger\_ = PLANNING\_R;

// get some parameters that will be global to the move base node

std::string global\_planner, local\_planner;//从参数服务器上读取配置的一些参数

private\_nh.param("base\_global\_planner", global\_planner, std::string("navfn/NavfnROS"));

private\_nh.param("base\_local\_planner", local\_planner, std::string("base\_local\_planner/TrajectoryPlannerROS"));

private\_nh.param("global\_costmap/robot\_base\_frame", robot\_base\_frame\_, std::string("base\_link"));

private\_nh.param("global\_costmap/global\_frame", global\_frame\_, std::string("map"));

private\_nh.param("planner\_frequency", planner\_frequency\_, 0.0);

private\_nh.param("controller\_frequency", controller\_frequency\_, 20.0);

private\_nh.param("planner\_patience", planner\_patience\_, 5.0);

private\_nh.param("controller\_patience", controller\_patience\_, 15.0);

private\_nh.param("max\_planning\_retries", max\_planning\_retries\_, -1); // disabled by default

private\_nh.param("oscillation\_timeout", oscillation\_timeout\_, 0.0);

private\_nh.param("oscillation\_distance", oscillation\_distance\_, 0.5);

// set up plan triple buffer

planner\_plan\_ = new std::vector<geometry\_msgs::PoseStamped>();

latest\_plan\_ = new std::vector<geometry\_msgs::PoseStamped>();

controller\_plan\_ = new std::vector<geometry\_msgs::PoseStamped>();

// set up the planner's thread 开启路径规划线程

planner\_thread\_ = new boost::thread(boost::bind(&MoveBase::planThread, this));

// for commanding the base

vel\_pub\_ = nh.advertise<geometry\_msgs::Twist>("cmd\_vel", 1);

current\_goal\_pub\_ = private\_nh.advertise<geometry\_msgs::PoseStamped>("current\_goal", 0);

ros::NodeHandle action\_nh("move\_base");

action\_goal\_pub\_ = action\_nh.advertise<move\_base\_msgs::MoveBaseActionGoal>("goal", 1);

// we'll provide a mechanism for some people to send goals as PoseStamped messages over a topic

// they won't get any useful information back about its status, but this is useful for tools

// like nav\_view and rviz

ros::NodeHandle simple\_nh("move\_base\_simple");

goal\_sub\_ = simple\_nh.subscribe<geometry\_msgs::PoseStamped>("goal", 1, boost::bind(&MoveBase::goalCB, this, \_1));

// we'll assume the radius of the robot to be consistent with what's specified for the costmaps

private\_nh.param("local\_costmap/inscribed\_radius", inscribed\_radius\_, 0.325);

private\_nh.param("local\_costmap/circumscribed\_radius", circumscribed\_radius\_, 0.46);

private\_nh.param("clearing\_radius", clearing\_radius\_, circumscribed\_radius\_);

private\_nh.param("conservative\_reset\_dist", conservative\_reset\_dist\_, 3.0);

private\_nh.param("shutdown\_costmaps", shutdown\_costmaps\_, false);

private\_nh.param("clearing\_rotation\_allowed", clearing\_rotation\_allowed\_, true);

private\_nh.param("recovery\_behavior\_enabled", recovery\_behavior\_enabled\_, true);

// create the ros wrapper for the planner's costmap... and initializer a pointer we'll use with the underlying map

planner\_costmap\_ros\_ = new costmap\_2d::Costmap2DROS("global\_costmap", tf\_);

planner\_costmap\_ros\_->pause();

// initialize the global planner

try

{

planner\_ = bgp\_loader\_.createInstance(global\_planner); // 加载libglobal\_planner.so动态库，获取到库的导出类，且创建该类的一个实例，利用父类智能指针指向它

planner\_->initialize(bgp\_loader\_.getName(global\_planner), planner\_costmap\_ros\_);

}

catch (const pluginlib::PluginlibException &ex)

{

ROS\_FATAL("Failed to create the %s planner, are you sure it is properly registered and that the containing library is built? Exception: %s", global\_planner.c\_str(), ex.what());

exit(1);

}

// create the ros wrapper for the controller's costmap... and initializer a pointer we'll use with the underlying map

controller\_costmap\_ros\_ = new costmap\_2d::Costmap2DROS("local\_costmap", tf\_);

controller\_costmap\_ros\_->pause();

// create a local planner

try

{

tc\_ = blp\_loader\_.createInstance(local\_planner);//加载libglocal\_planner.so动态库，获取到库的导出类，且创建该类的一个实例，利用父类智能指针指向它

ROS\_INFO("Created local\_planner %s", local\_planner.c\_str());

tc\_->initialize(blp\_loader\_.getName(local\_planner), &tf\_, controller\_costmap\_ros\_);

}

catch (const pluginlib::PluginlibException &ex)

{

ROS\_FATAL("Failed to create the %s planner, are you sure it is properly registered and that the containing library is built? Exception: %s", local\_planner.c\_str(), ex.what());

exit(1);

}

// Start actively updating costmaps based on sensor data开始更新这两种代价地图

planner\_costmap\_ros\_->start();

controller\_costmap\_ros\_->start();

// advertise a service for getting a plan//分别定义路径规划的服务和清除代价地图的服务，回调函数分别是planservice和clearcostmapService

make\_plan\_srv\_ = private\_nh.advertiseService("make\_plan", &MoveBase::planService, this);

// advertise a service for clearing the costmaps

clear\_costmaps\_srv\_ = private\_nh.advertiseService("clear\_costmaps", &MoveBase::clearCostmapsService, this);

// if we shutdown our costmaps when we're deactivated... we'll do that now

if (shutdown\_costmaps\_)

{

ROS\_DEBUG\_NAMED("move\_base", "Stopping costmaps initially");

planner\_costmap\_ros\_->stop();

controller\_costmap\_ros\_->stop();

}

// load any user specified recovery behaviors, and if that fails load the defaults

if (!loadRecoveryBehaviors(private\_nh))

{

loadDefaultRecoveryBehaviors();

}

// initially, we'll need to make a plan

state\_ = PLANNING;

// we'll start executing recovery behaviors at the beginning of our list

recovery\_index\_ = 0;

// we're all set up now so we can start the action server

as\_->start(); // 配置完毕，启动路径规划线程,这里开启的是用于局部规划的executeCb线程

// 动态参数服务器设置

dsrv\_ = new dynamic\_reconfigure::Server<move\_base::MoveBaseConfig>(ros::NodeHandle("~"));

dynamic\_reconfigure::Server<move\_base::MoveBaseConfig>::CallbackType cb = boost::bind(&MoveBase::reconfigureCB, this, \_1, \_2);

dsrv\_->setCallback(cb);

}

注：

1）// set up the planner's thread 开启路径规划线程

planner\_thread\_ = new boost::thread(boost::bind(&MoveBase::planThread, this));

2）as\_ = new MoveBaseActionServer(ros::NodeHandle(), "move\_base", boost::bind(&MoveBase::executeCb, this, \_1), false);

// as\_维护movebase的actionServer状态机，并且新建了一个executeCb线程。as\_->start(); // 配置完毕，启动路径规划线程,这里开启的是用于局部规划的executeCb线程

1. planThread()

void MoveBase::planThread()

{

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "Starting planner thread...");

ros::NodeHandle n;

ros::Timer timer;

bool wait\_for\_wake = false;

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);

ros::Time deal\_clear = ros::Time::now();

while (n.ok())//需要不断执行

{

if ((ros::Time::now() - deal\_clear).toSec() > 3.0)

{

this->clearCostmaps();

deal\_clear = ros::Time::now();

}

// check if we should run the planner (the mutex is locked)检查是否需要进行planner

//条件变量阻塞线程，利用生产消费者模型，如果需要等待，或者在局部规划时不需要planner，进入while循环，

//并悬挂当前线程。planner\_cond\_.wait(lock);会使得线程悬挂，线程阻塞在这，等待其他线程planner\_cond\_.notify\_one();指令来

//唤醒线程。

while (wait\_for\_wake || !runPlanner\_)

{ // 等待

// if we should not be running the planner then suspend this thread

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "Planner thread is suspending");

planner\_cond\_.wait(lock);

wait\_for\_wake = false;

}

ros::Time start\_time = ros::Time::now();

// time to plan! get a copy of the goal and unlock the mutex

geometry\_msgs::PoseStamped temp\_goal = planner\_goal\_;

lock.unlock();

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "Planning...");

// run planner

planner\_plan\_->clear();

bool gotPlan = n.ok() && makePlan(temp\_goal, \*planner\_plan\_); // 规划全局路径，放在planner\_plan\_地址

if (gotPlan)//如果规划成功

{

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "Got Plan with %zu points!", planner\_plan\_->size());

// pointer swap the plans under mutex (the controller will pull from latest\_plan\_)

std::vector<geometry\_msgs::PoseStamped> \*temp\_plan = planner\_plan\_;

lock.lock();//上锁，保护内部变量，防止其他线程操作

planner\_plan\_ = latest\_plan\_;

latest\_plan\_ = temp\_plan;//更新最新的全局规划结果

last\_valid\_plan\_ = ros::Time::now();

planning\_retries\_ = 0;//规划次数重置

new\_global\_plan\_ = true;//后面如果有了新的规划，按照新的规划进行局部规划

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "Generated a plan from the base\_global\_planner");

// make sure we only start the controller if we still haven't reached the goal

if (runPlanner\_)

state\_ = CONTROLLING;//切换状态

if (planner\_frequency\_ <= 0)

runPlanner\_ = false;//全局规划完成，悬挂全局规划线程，等待新的终点指令

lock.unlock();

}

// if we didn't get a plan and we are in the planning state (the robot isn't moving)

else if (state\_ == PLANNING)//如果规划没有得出结果

{

ROS\_DEBUG\_NAMED("move\_base\_plan\_thread", "No Plan...");

ros::Time attempt\_end = last\_valid\_plan\_ + ros::Duration(planner\_patience\_);//本来规划需要的最大时间

// check if we've tried to make a plan for over our time limit or our maximum number of retries

// issue #496: we stop planning when one of the conditions is true, but if max\_planning\_retries\_

// is negative (the default), it is just ignored and we have the same behavior as ever

lock.lock();//加锁，保护内部变量不被其他线程修改

planning\_retries\_++;//规划次数加1

if (runPlanner\_ &&

(ros::Time::now() > attempt\_end || planning\_retries\_ > uint32\_t(max\_planning\_retries\_)))//规划超时，次数超过最大尝试次数

{

// we'll move into our obstacle clearing mode

state\_ = CLEARING;

runPlanner\_ = false; // proper solution for issue #523悬挂全局规划

publishZeroVelocity();

recovery\_trigger\_ = PLANNING\_R;

}

lock.unlock();

}

// take the mutex for the next iteration

lock.lock();

// setup sleep interface if needed

if (planner\_frequency\_ > 0)

{//start\_time + ros::Duration(1.0 / planner\_frequency\_)为本来规划完需要的时间，减去ros::Time::now()为超时多少

ros::Duration sleep\_time = (start\_time + ros::Duration(1.0 / planner\_frequency\_)) - ros::Time::now();

if (sleep\_time > ros::Duration(0.0))

{

wait\_for\_wake = true;

timer = n.createTimer(sleep\_time, &MoveBase::wakePlanner, this);

}

}

}

}

其中:

1)这个线程一开始就要一个while (n.ok())函数循环，一直执行规划任务，不需要规划时挂起即可。

2)makePlan(temp\_goal, \*planner\_plan\_)是具体的全局规划函数，如下

bool MoveBase::makePlan(const geometry\_msgs::PoseStamped &goal, std::vector<geometry\_msgs::PoseStamped> &plan)

{

boost::unique\_lock<costmap\_2d::Costmap2D::mutex\_t> lock(\*(planner\_costmap\_ros\_->getCostmap()->getMutex()));

// make sure to set the plan to be empty initially

plan.clear();//清空数组

// since this gets called on handle activate

if (planner\_costmap\_ros\_ == NULL)

{

ROS\_ERROR("Planner costmap ROS is NULL, unable to create global plan");

return false;

}

// get the starting pose of the robot 获取小车全局位置，在代价地图中

geometry\_msgs::PoseStamped global\_pose;

if (!getRobotPose(global\_pose, planner\_costmap\_ros\_))

{

ROS\_WARN("Unable to get starting pose of robot, unable to create global plan");

return false;

}

const geometry\_msgs::PoseStamped &start = global\_pose;

// if the planner fails or returns a zero length plan, planning failed

if (!planner\_->makePlan(start, goal, plan) || plan.empty())//父类智能指针planner\_利用多态调用global\_planner子类makePlan(start, goal, plan)函数。

{

ROS\_DEBUG\_NAMED("move\_base", "Failed to find a plan to point (%.2f, %.2f)", goal.pose.position.x, goal.pose.position.y);

return false;

}

return true;

}

这里，父类指针利用多态调用了全局规划子类的成员函数计算全局路径点，如下：



1. executeCb()

void MoveBase::executeCb(const move\_base\_msgs::MoveBaseGoalConstPtr &move\_base\_goal)

{

if (!isQuaternionValid(move\_base\_goal->target\_pose.pose.orientation))//检查四元素是否有效

{

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Aborting on goal because it was sent with an invalid quaternion");

return;

}

geometry\_msgs::PoseStamped goal = goalToGlobalFrame(move\_base\_goal->target\_pose);//转换到全局坐标系

// we have a goal so start the planner

// 上锁，其他线程想要拿到这把锁时，无法拿到，会阻塞在那个位置，为了保护planner\_goal\_ = goal完全执行

//planner\_goal\_ = goal完全执行完后，唤醒线程，并runPlanner\_ = true;开始规划

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);

planner\_goal\_ = goal;

runPlanner\_ = true;//确认需要规划

planner\_cond\_.notify\_one();//唤醒线程

lock.unlock();

current\_goal\_pub\_.publish(goal);//发布当前位置话题

std::vector<geometry\_msgs::PoseStamped> global\_plan;

ros::Rate r(controller\_frequency\_);

if (shutdown\_costmaps\_)

{

ROS\_DEBUG\_NAMED("move\_base", "Starting up costmaps that were shut down previously");

planner\_costmap\_ros\_->start();

controller\_costmap\_ros\_->start();

}

// we want to make sure that we reset the last time we had a valid plan and control

last\_valid\_control\_ = ros::Time::now();

last\_valid\_plan\_ = ros::Time::now();

last\_oscillation\_reset\_ = ros::Time::now();

planning\_retries\_ = 0;

ros::NodeHandle n;//创建节点句柄

while (n.ok())

{

if (c\_freq\_change\_)//如果打算局部规划频率

{

ROS\_INFO("Setting controller frequency to %.2f", controller\_frequency\_);

r = ros::Rate(controller\_frequency\_);//局部规划频率设置为

c\_freq\_change\_ = false;

}

if (as\_->isPreemptRequested())

{

if (as\_->isNewGoalAvailable())

{ // 如果接收到新的目标点

// if we're active and a new goal is available, we'll accept it, but we won't shut anything down

move\_base\_msgs::MoveBaseGoal new\_goal = \*as\_->acceptNewGoal();

if (!isQuaternionValid(new\_goal.target\_pose.pose.orientation))

{ // 判断位置有效性

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Aborting on goal because it was sent with an invalid quaternion");

return;

}

goal = goalToGlobalFrame(new\_goal.target\_pose); // 将位置坐标转化为全局

// we'll make sure that we reset our state for the next execution cycle

recovery\_index\_ = 0;

state\_ = PLANNING; // 重新更新状态，设置为规划状态，此时控制模式暂时停止

// we have a new goal so make sure the planner is awake

lock.lock(); // 开启规划线程锁，进行全局规划

planner\_goal\_ = goal;

runPlanner\_ = true;

planner\_cond\_.notify\_one();

lock.unlock();

// publish the goal point to the visualizer

ROS\_DEBUG\_NAMED("move\_base", "move\_base has received a goal of x: %.2f, y: %.2f", goal.pose.position.x, goal.pose.position.y);

current\_goal\_pub\_.publish(goal); // 发布当前位置

// make sure to reset our timeouts and counters

last\_valid\_control\_ = ros::Time::now(); // 更新时间

last\_valid\_plan\_ = ros::Time::now();

last\_oscillation\_reset\_ = ros::Time::now();

planning\_retries\_ = 0;

}

else

{

// if we've been preempted explicitly we need to shut things down

resetState();

// notify the ActionServer that we've successfully preempted

ROS\_DEBUG\_NAMED("move\_base", "Move base preempting the current goal");

as\_->setPreempted();

// we'll actually return from execute after preempting

return;

}

}

// we also want to check if we've changed global frames because we need to transform our goal pose

if (goal.header.frame\_id != planner\_costmap\_ros\_->getGlobalFrameID())//是否重新指定了新goal

{

goal = goalToGlobalFrame(goal);//更新新目标点

// we want to go back to the planning state for the next execution cycle

recovery\_index\_ = 0;

state\_ = PLANNING;

// we have a new goal so make sure the planner is awake

lock.lock();

planner\_goal\_ = goal;

runPlanner\_ = true;

planner\_cond\_.notify\_one();

lock.unlock();

// publish the goal point to the visualizer

ROS\_DEBUG\_NAMED("move\_base", "The global frame for move\_base has changed, new frame: %s, new goal position x: %.2f, y: %.2f", goal.header.frame\_id.c\_str(), goal.pose.position.x, goal.pose.position.y);

current\_goal\_pub\_.publish(goal);

// make sure to reset our timeouts and counters

last\_valid\_control\_ = ros::Time::now();

last\_valid\_plan\_ = ros::Time::now();

last\_oscillation\_reset\_ = ros::Time::now();

planning\_retries\_ = 0;

}

// for timing that gives real time even in simulation

ros::WallTime start = ros::WallTime::now();

// the real work on pursuing a goal is done here

bool done = executeCycle(goal, global\_plan); // 进行局部路径规划，如果没有到达目的地，一直返回false，进行下一次循环计算控制指令

// if we're done, then we'll return from execute

if (done)

return;//如果到达目的地，终止导航

// check if execution of the goal has completed in some way

ros::WallDuration t\_diff = ros::WallTime::now() - start;

ROS\_DEBUG\_NAMED("move\_base", "Full control cycle time: %.9f\n", t\_diff.toSec());//输出当前控制指令计算消耗时间的调试信息

r.sleep();

// make sure to sleep for the remainder of our cycle time

if (r.cycleTime() > ros::Duration(1 / controller\_frequency\_) && state\_ == CONTROLLING)//控制计算超时

ROS\_WARN("Control loop missed its desired rate of %.4fHz... the loop actually took %.4f seconds", controller\_frequency\_, r.cycleTime().toSec());

}

// wake up the planner thread so that it can exit cleanly

lock.lock();

runPlanner\_ = true;

planner\_cond\_.notify\_one();

lock.unlock();

// if the node is killed then we'll abort and return

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Aborting on the goal because the node has been killed");

return;

}

其中：

1. 以下函数会调用局部规划子类的c\_->computeVelocityCommands(cmd\_vel)函数计算控制指令，并利用vel\_pub\_.publish(cmd\_vel);将控制指令以话题的形式发布出去。



1. executeCycle()函数如下：

bool MoveBase::executeCycle(geometry\_msgs::PoseStamped &goal, std::vector<geometry\_msgs::PoseStamped> &global\_plan)

{

boost::recursive\_mutex::scoped\_lock ecl(configuration\_mutex\_);//加锁，其他线程如果运行到这把锁，会被阻塞，直到这里解锁

// we need to be able to publish velocity commands

geometry\_msgs::Twist cmd\_vel;//实例化控制指令消息

// update feedback to correspond to our curent position

geometry\_msgs::PoseStamped global\_pose;

getRobotPose(global\_pose, planner\_costmap\_ros\_);

const geometry\_msgs::PoseStamped &current\_position = global\_pose;

// push the feedback out

move\_base\_msgs::MoveBaseFeedback feedback;

feedback.base\_position = current\_position;

as\_->publishFeedback(feedback);

// check to see if we've moved far enough to reset our oscillation timeout

if (distance(current\_position, oscillation\_pose\_) >= oscillation\_distance\_)

{

last\_oscillation\_reset\_ = ros::Time::now();

oscillation\_pose\_ = current\_position;

// if our last recovery was caused by oscillation, we want to reset the recovery index

if (recovery\_trigger\_ == OSCILLATION\_R)

recovery\_index\_ = 0;

}

// check that the observation buffers for the costmap are current, we don't want to drive blind

if (!controller\_costmap\_ros\_->isCurrent())//判断传感器数据是否最新

{

ROS\_WARN("[%s]:Sensor data is out of date, we're not going to allow commanding of the base for safety", ros::this\_node::getName().c\_str());

publishZeroVelocity();

return false;

}

// if we have a new plan then grab it and give it to the controller

if (new\_global\_plan\_)//如果有了一个新的全局规划

{

// make sure to set the new plan flag to false

new\_global\_plan\_ = false;

ROS\_DEBUG\_NAMED("move\_base", "Got a new plan...swap pointers");

// do a pointer swap under mutex

std::vector<geometry\_msgs::PoseStamped> \*temp\_plan = controller\_plan\_;

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);

controller\_plan\_ = latest\_plan\_;//设置全局规划规划路径为controller\_plan\_进行局部规划

latest\_plan\_ = temp\_plan;//上一个全局规划路径保存下来

lock.unlock();

ROS\_DEBUG\_NAMED("move\_base", "pointers swapped!");

if (!tc\_->setPlan(\*controller\_plan\_))//重新设置全局规划路径为controller\_plan\_进行局部规划

{

// ABORT and SHUTDOWN COSTMAPS

ROS\_ERROR("Failed to pass global plan to the controller, aborting.");

resetState();

// disable the planner thread

lock.lock();

runPlanner\_ = false;

lock.unlock();

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Failed to pass global plan to the controller.");

return true;

}

// make sure to reset recovery\_index\_ since we were able to find a valid plan

if (recovery\_trigger\_ == PLANNING\_R)

recovery\_index\_ = 0;

}

// the move\_base state machine, handles the control logic for navigation

switch (state\_)//根据状态来选择执行全局规划，局部规划，或者是清理障碍

{

// if we are in a planning state, then we'll attempt to make a plan

case PLANNING:

{

boost::recursive\_mutex::scoped\_lock lock(planner\_mutex\_);

runPlanner\_ = true;

planner\_cond\_.notify\_one();

}

ROS\_DEBUG\_NAMED("move\_base", "Waiting for plan, in the planning state.");

break;

// if we're controlling, we'll attempt to find valid velocity commands

case CONTROLLING:

ROS\_DEBUG\_NAMED("move\_base", "In controlling state.");

// check to see if we've reached our goal

if (tc\_->isGoalReached())//先检查是否到达目的地

{

/\*ROS\_DEBUG\_NAMED("move\_base","Goal reached!");\*/

/\*ROS\_INFO("move\_base","Goal reached!");\*/

ROS\_DEBUG\_NAMED("move\_base", "Goal reached!");

ROS\_INFO("Goal reached!");

resetState();//到达目的地后重置状态

// disable the planner thread

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);//上锁，保护变量runPlanner\_赋值，防止其他线程争抢赋值

runPlanner\_ = false;

lock.unlock();

as\_->setSucceeded(move\_base\_msgs::MoveBaseResult(), "Goal reached.");

return true;

}

// check for an oscillation condition

if (oscillation\_timeout\_ > 0.0 &&

last\_oscillation\_reset\_ + ros::Duration(oscillation\_timeout\_) < ros::Time::now())

{

publishZeroVelocity();

state\_ = CLEARING;

recovery\_trigger\_ = OSCILLATION\_R;

}

{//进行控制指令计算，会上锁

boost::unique\_lock<costmap\_2d::Costmap2D::mutex\_t> lock(\*(controller\_costmap\_ros\_->getCostmap()->getMutex()));

if (tc\_->computeVelocityCommands(cmd\_vel))

{ // 调用局部规划器，computeVelocityCommands函数是teb整个规划器的函数接口，里面还有函数planner\_->plan（）（这个函数调用图优化）。

// 具体computeVelocityCommands函数过程是先初始化，然后调用planner\_->plan（）规划局部路径，然后再计算速度，获得cmd\_vel

ROS\_DEBUG\_NAMED("move\_base", "Got a valid command from the local planner: %.3lf, %.3lf, %.3lf",

cmd\_vel.linear.x, cmd\_vel.linear.y, cmd\_vel.angular.z);

last\_valid\_control\_ = ros::Time::now();

// make sure that we send the velocity command to the base

vel\_pub\_.publish(cmd\_vel);//发布控制指令

if (recovery\_trigger\_ == CONTROLLING\_R)

recovery\_index\_ = 0;

}

else

{

ROS\_DEBUG\_NAMED("move\_base", "The local planner could not find a valid plan.");

ros::Time attempt\_end = last\_valid\_control\_ + ros::Duration(controller\_patience\_);

// check if we've tried to find a valid control for longer than our time limit

if (ros::Time::now() > attempt\_end)//如果局部规划超时

{

// we'll move into our obstacle clearing mode

publishZeroVelocity();

state\_ = CLEARING;//切换状态为清理模式

recovery\_trigger\_ = CONTROLLING\_R;

}

else

{

// otherwise, if we can't find a valid control, we'll go back to planning

//如果无法获得有效的控制指令，会切换状态为PLANNING，重新规划一条全局路径

last\_valid\_plan\_ = ros::Time::now();

planning\_retries\_ = 0;

state\_ = PLANNING;

publishZeroVelocity();

// enable the planner thread in case it isn't running on a clock

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);//唤醒全局规划线程

runPlanner\_ = true;

planner\_cond\_.notify\_one();

lock.unlock();

}

}

}

break;

// we'll try to clear out space with any user-provided recovery behaviors

case CLEARING:

ROS\_DEBUG\_NAMED("move\_base", "In clearing/recovery state");

// we'll invoke whatever recovery behavior we're currently on if they're enabled

if (recovery\_behavior\_enabled\_ && recovery\_index\_ < recovery\_behaviors\_.size())

{

ROS\_DEBUG\_NAMED("move\_base\_recovery", "Executing behavior %u of %zu", recovery\_index\_, recovery\_behaviors\_.size());

recovery\_behaviors\_[recovery\_index\_]->runBehavior();

// we at least want to give the robot some time to stop oscillating after executing the behavior

last\_oscillation\_reset\_ = ros::Time::now();

// we'll check if the recovery behavior actually worked

ROS\_DEBUG\_NAMED("move\_base\_recovery", "Going back to planning state");

last\_valid\_plan\_ = ros::Time::now();

planning\_retries\_ = 0;

state\_ = PLANNING;

// update the index of the next recovery behavior that we'll try

recovery\_index\_++;

}

else

{

ROS\_DEBUG\_NAMED("move\_base\_recovery", "All recovery behaviors have failed, locking the planner and disabling it.");

// disable the planner thread

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);

runPlanner\_ = false;

lock.unlock();

ROS\_DEBUG\_NAMED("move\_base\_recovery", "Something should abort after this.");

if (recovery\_trigger\_ == CONTROLLING\_R)

{

ROS\_ERROR("Aborting because a valid control could not be found. Even after executing all recovery behaviors");

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Failed to find a valid control. Even after executing recovery behaviors.");

}

else if (recovery\_trigger\_ == PLANNING\_R)

{

ROS\_ERROR("Aborting because a valid plan could not be found. Even after executing all recovery behaviors");

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Failed to find a valid plan. Even after executing recovery behaviors.");

}

else if (recovery\_trigger\_ == OSCILLATION\_R)

{

ROS\_ERROR("Aborting because the robot appears to be oscillating over and over. Even after executing all recovery behaviors");

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Robot is oscillating. Even after executing recovery behaviors.");

}

resetState();

return true;

}

break;

default:

ROS\_ERROR("This case should never be reached, something is wrong, aborting");

resetState();

// disable the planner thread

boost::unique\_lock<boost::recursive\_mutex> lock(planner\_mutex\_);

runPlanner\_ = false;

lock.unlock();

as\_->setAborted(move\_base\_msgs::MoveBaseResult(), "Reached a case that should not be hit in move\_base. This is a bug, please report it.");

return true;

}

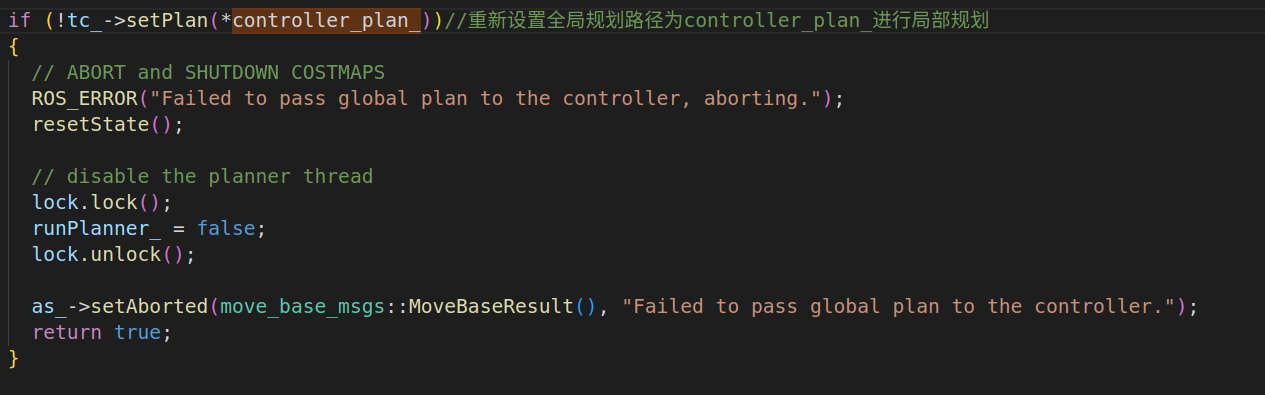
// we aren't done yet

return false;//正常规划情况下，如果还没到达目标点，会一直返回false

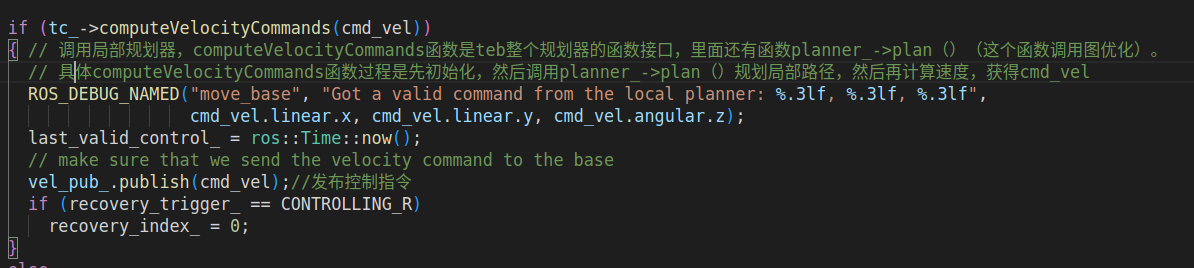
}

其中：

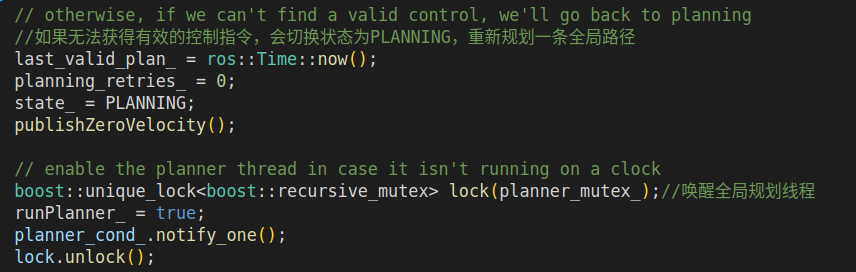
1. 如果重新设置了新目标点，有了新的全局规划后，重新设置全局规划为控制规划进行局部规划



1. 调用子类的tc\_->computeVelocityCommands(cmd\_vel)函数计算控制指令,并利用vel\_pub\_.publish(cmd\_vel);//发布控制指令，是引用传递的方式。



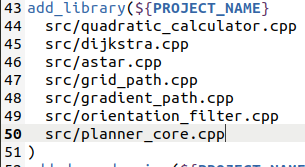
1. 如果长期无法获得控制指令会重新进行全局规划



接下来分析全局和局部规划的具体函数：

1. 具体的全局规划函数planner\_->makePlan(start, goal, plan) 重要说明如下：

4.1查找动态库发现其主要的cpp文件主要是以下这些：



planner\_智能指针指向的子类就在planner\_core.h和planner\_core.cpp文件中定义和解释。

4.2 makePlan()函数内部主要包括以下：

4.2.1 在头文件中创建全局规划器父类指针

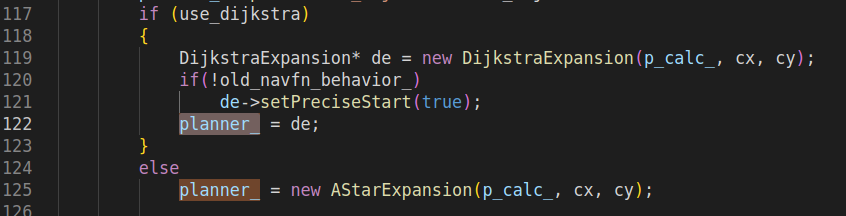


前者在初始化函数中父类指针根据参数服务器中的参数来选择子类全局规划器，并指向

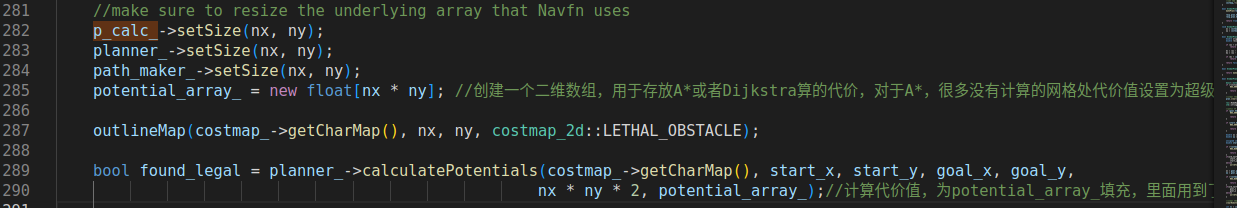
首先获取服务器参数：



然后选择

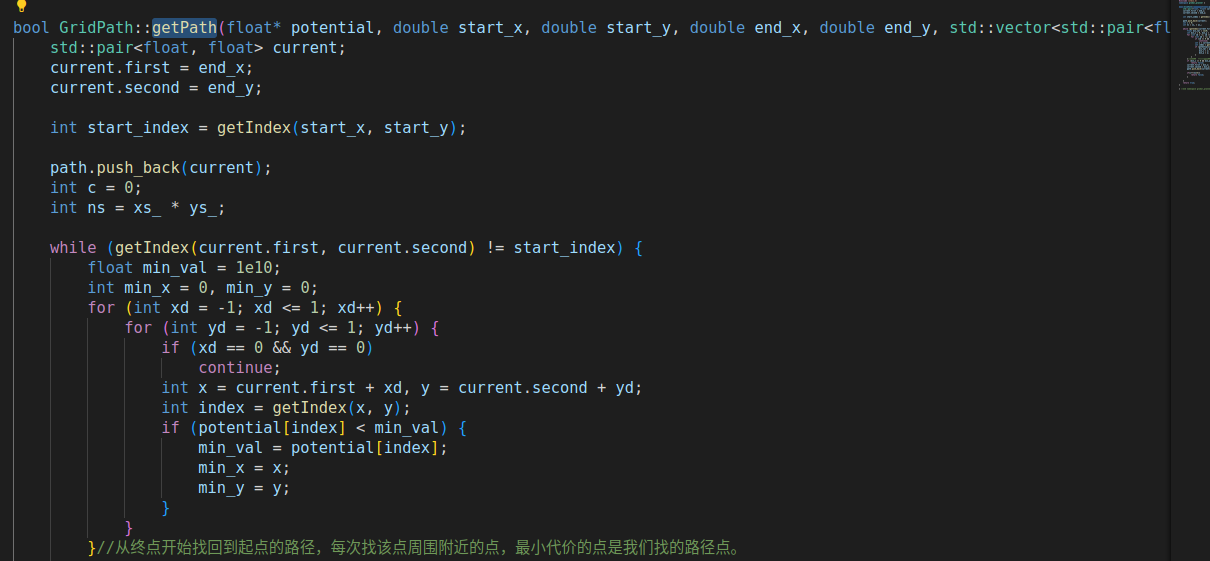


4.2.2 planner\_->calculatePotentials(costmap\_->getCharMap(), start\_x, start\_y, goal\_x, goal\_y,nx \* ny \* 2, potential\_array\_)利用A\*或者Dijkstra方法计算全局代价值。将结果放在potential\_array\_数组里面，这里的数组存放了二维代价map，数组元素的位置需要和二维代价map做一个映射。

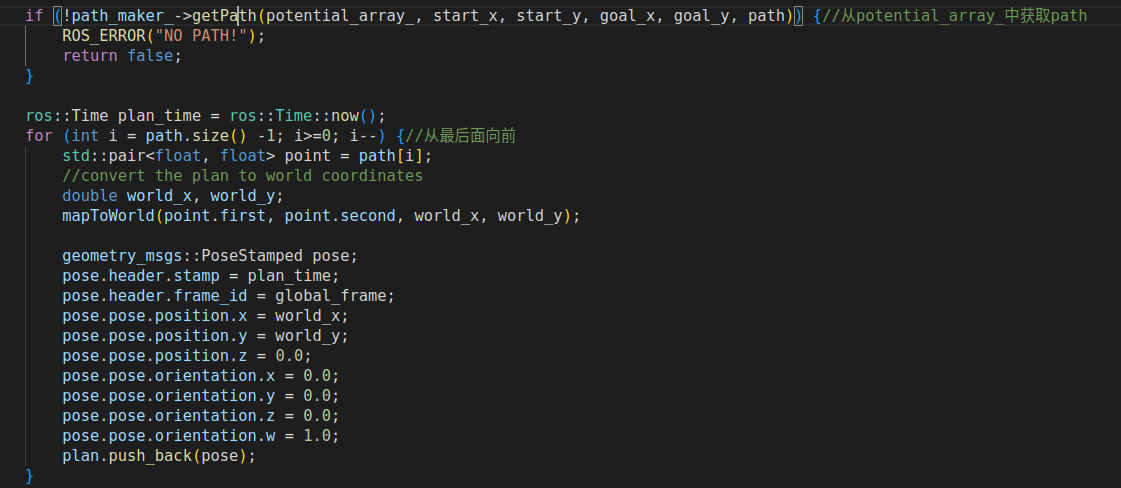


4.3 获得potential\_array\_数组后，利用getPlanFromPotential(start\_x, start\_y, goal\_x, goal\_y, goal, plan)根据代价获取全局路径点，具体如下：

4.3.1首先从最后一个路径点开始，遍历周围8个点的代价值，并将代价值最小点的x和y位置坐标放进path动态数组中，并更新当前点



4.3.2然后取出path的数据，转化到全局坐标系，然后填充pose，push\_back进入plan动态数组中，得到规划路径。



4.3.3然后plan为指针传入，所以planner\_->makePlan(start, goal, plan) 获得全局路径。

4.4 全局路径规划算法A\*求解代价数组分析如下：

bool AStarExpansion::calculatePotentials(unsigned char\* costs, double start\_x, double start\_y, double end\_x, double end\_y,

int cycles, float\* potential) {

queue\_.clear();

//动态数组queue\_ 用于加入周围可行点，且寻找最小代价值，以更新

int start\_i = toIndex(start\_x, start\_y);//坐标对应的potential数组下标

queue\_.push\_back(Index(start\_i, 0));

std::fill(potential, potential + ns\_, POT\_HIGH);

potential[start\_i] = 0;

int goal\_i = toIndex(end\_x, end\_y);

int cycle = 0;

while (queue\_.size() > 0 && cycle < cycles) { //以当前点为中心，向4个方向计算代价值

Index top = queue\_[0]; //获取代价值小的元素

//pop\_heap是在堆的基础上，弹出堆顶元素，需要注意的是，pop\_heap()并没有删除元素，而是将堆顶元素和数组最后一个元素进行了替换，如果要删除这个元素，还需要对数组进行pop\_back()操作

std::pop\_heap(queue\_.begin(), queue\_.end(), greater1());

queue\_.pop\_back();//弹出堆顶元素

int i = top.i;

if (i == goal\_i)

return true;//如果到达目标点，返回

//分别将上下左右相邻网格点计算代价值，并加入queue\_

add(costs, potential, potential[i], i + 1, end\_x, end\_y);

add(costs, potential, potential[i], i - 1, end\_x, end\_y);

add(costs, potential, potential[i], i + nx\_, end\_x, end\_y);

add(costs, potential, potential[i], i - nx\_, end\_x, end\_y);

cycle++;

}

return false;

}

void AStarExpansion::add(unsigned char\* costs, float\* potential, float prev\_potential, int next\_i, int end\_x,

int end\_y) {

if (next\_i < 0 || next\_i >= ns\_)

return;

if (potential[next\_i] < POT\_HIGH)//如果该点在障碍物上，直接返回

return;

if(costs[next\_i]>=lethal\_cost\_ && !(unknown\_ && costs[next\_i]==costmap\_2d::NO\_INFORMATION))

return;

potential[next\_i] = p\_calc\_->calculatePotential(potential, costs[next\_i] + neutral\_cost\_, next\_i, prev\_potential);

int x = next\_i % nx\_, y = next\_i / nx\_;

float distance = abs(end\_x - x) + abs(end\_y - y);

queue\_.push\_back(Index(next\_i, potential[next\_i] + distance \* neutral\_cost\_));//先计算综合代价，再push进入queue\_

std::push\_heap(queue\_.begin(), queue\_.end(), greater1());//push\_heap带greater1()使得queue\_数据进行小顶堆排序

}

virtual float calculatePotential(float\* potential, unsigned char cost, int n, float prev\_potential=-1){

if(prev\_potential < 0){

// get min of neighbors

float min\_h = std::min( potential[n - 1], potential[n + 1] ),

min\_v = std::min( potential[n - nx\_], potential[n + nx\_]);

prev\_potential = std::min(min\_h, min\_v);

}

return prev\_potential + cost;

}

当前点的potential 等于 前后左右最小的potential + 当前的costs

4.5 全局路径规划算法Dijkstra求解代价数组分析如下：

参考：

<https://blog.csdn.net/qq825255961/article/details/114840555?ops_request_misc=&request_id=&biz_id=102&utm_term=DijkstraExpansion::calculatePo&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-114840555.142^v83^koosearch_v1,239^v2^insert_chatgpt&spm=1018.2226.3001.4187>

主要流程是首先初始化，将起始点四周的点加入优先级队列，然后进入主循环

主循环先判断当前优先级队列是否为空，如果为空则代表没办法扩展了直接退出return false

记录一下最大优先级队列大小

将当前优先级队列的pending全部设为false

重新获取当前队列的指针和长度

对当前队列每个点调用updatecell函数，计算每个点周围点的代价值并根据需要写入下一队列

调用完成之后，交换当前队列与下一队列

查看当前队列是否为空，空的话代表没有什么可以扩展的了

如果为空则调大临界点的阈值（最开始是致命代价），将关闭列表与当前列表互换，进行下一次扩展（这里表明了无论是否大于致命代价，地杰斯特拉算法总会扩展到目标点）

判断目标点是否被扩展到，如果扩展到了就退出

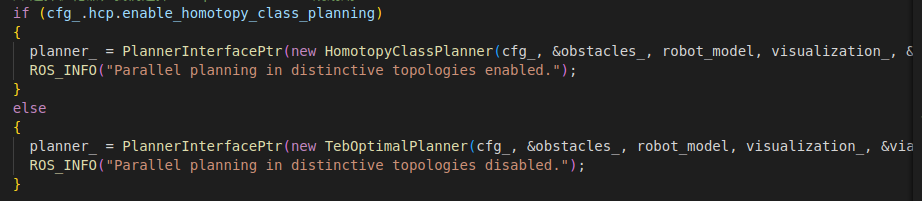
1. 具体的局部规划函数tc\_->computeVelocityCommands(cmd\_vel)重要说明如下：

TEB局部规划源码分析如下：

* 1. 首先利用tc\_->initialize(blp\_loader\_.getName(local\_planner), &tf\_, controller\_costmap\_ros\_);对TebLocalPlannerROS类成员进行初始化。

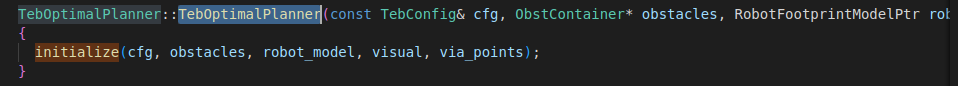
主要包括：

5.1.1选择优化器，我们选择TebOptimalPlanner规划期



其中：

会进行有参构造：内部还会进行初始化



后面再讲TebOptimalPlanner类的成员和成员函数。

5.1.2代价地图赋值给内部成员：



5.1.3加载参数，放在cfg\_对象内部：



* 1. 将需要规划的全局路径放进tc\_所指向的对象



* 1. 然后正式进入tc\_->computeVelocityCommands(cmd\_vel),内部再调用以下函数

uint32\_t TebLocalPlannerROS::computeVelocityCommands(const geometry\_msgs::PoseStamped& pose,

const geometry\_msgs::TwistStamped& velocity,

geometry\_msgs::TwistStamped &cmd\_vel,

std::string &message)

{

// check if plugin initialized

if(!initialized\_) //初始化

{

ROS\_ERROR("teb\_local\_planner has not been initialized, please call initialize() before using this planner");

message = "teb\_local\_planner has not been initialized";

return mbf\_msgs::ExePathResult::NOT\_INITIALIZED;

}

static uint32\_t seq = 0;

cmd\_vel.header.seq = seq++;

cmd\_vel.header.stamp = ros::Time::now();

cmd\_vel.header.frame\_id = robot\_base\_frame\_;

cmd\_vel.twist.linear.x = cmd\_vel.twist.linear.y = cmd\_vel.twist.angular.z = 0;

goal\_reached\_ = false;

// Get robot pose// 获得机器人位姿

geometry\_msgs::PoseStamped robot\_pose;

costmap\_ros\_->getRobotPose(robot\_pose);//引用传入，填充robot\_pose

robot\_pose\_ = PoseSE2(robot\_pose.pose);//写到成员变量里面//PoseSE2内部有成员Eigen::Vector2d \_position; 和double \_theta;

// Get robot velocity// 获得机器人速度

geometry\_msgs::PoseStamped robot\_vel\_tf;

odom\_helper\_.getRobotVel(robot\_vel\_tf);//引用传递，将里程计信息给到robot\_vel\_tf

robot\_vel\_.linear.x = robot\_vel\_tf.pose.position.x;

robot\_vel\_.linear.y = robot\_vel\_tf.pose.position.y;

robot\_vel\_.angular.z = tf2::getYaw(robot\_vel\_tf.pose.orientation);//四元素到欧拉角转换

// prune global plan to cut off parts of the past (spatially before the robot)// 裁剪已经走过的全局路径 (spatially before the robot)

pruneGlobalPlan(\*tf\_, robot\_pose, global\_plan\_, cfg\_.trajectory.global\_plan\_prune\_distance);

// Transform global plan to the frame of interest (w.r.t. the local costmap)

std::vector<geometry\_msgs::PoseStamped> transformed\_plan;

int goal\_idx;

geometry\_msgs::TransformStamped tf\_plan\_to\_global;

if (!transformGlobalPlan(\*tf\_, global\_plan\_, robot\_pose, \*costmap\_, global\_frame\_, cfg\_.trajectory.max\_global\_plan\_lookahead\_dist,

transformed\_plan, &goal\_idx, &tf\_plan\_to\_global))//根据参数截取一段全局路径并做坐标转换

{

ROS\_WARN("Could not transform the global plan to the frame of the controller");

message = "Could not transform the global plan to the frame of the controller";

return mbf\_msgs::ExePathResult::INTERNAL\_ERROR;

}

// update via-points container

if (!custom\_via\_points\_active\_)

updateViaPointsContainer(transformed\_plan, cfg\_.trajectory.global\_plan\_viapoint\_sep);

// check if global goal is reached// 检查是否已经到了目标点

geometry\_msgs::PoseStamped global\_goal;

tf2::doTransform(global\_plan\_.back(), global\_goal, tf\_plan\_to\_global);

double dx = global\_goal.pose.position.x - robot\_pose\_.x();

double dy = global\_goal.pose.position.y - robot\_pose\_.y();

double delta\_orient = g2o::normalize\_theta( tf2::getYaw(global\_goal.pose.orientation) - robot\_pose\_.theta() );//计算角度偏差，tf2::getYaw(global\_goal.pose.orientation)是将四元素转化为欧拉角

if(fabs(std::sqrt(dx\*dx+dy\*dy)) < cfg\_.goal\_tolerance.xy\_goal\_tolerance

&& fabs(delta\_orient) < cfg\_.goal\_tolerance.yaw\_goal\_tolerance

&& (!cfg\_.goal\_tolerance.complete\_global\_plan || via\_points\_.size() == 0))//判断是否再阈值内

{

goal\_reached\_ = true;

return mbf\_msgs::ExePathResult::SUCCESS;

}

// check if we should enter any backup mode and apply settings // 检查是否进入备份模式并进行相关的设置

configureBackupModes(transformed\_plan, goal\_idx);

// Return false if the transformed global plan is empty // 如果转换后的全局路径为空，返回false

if (transformed\_plan.empty())

{

ROS\_WARN("Transformed plan is empty. Cannot determine a local plan.");

message = "Transformed plan is empty";

return mbf\_msgs::ExePathResult::INVALID\_PATH;

}

// Get current goal point (last point of the transformed plan)// 获得当前的目标点，也就是transformed\_plan的最后一个点，transformed\_plan是当前这一段的全局规划路径

robot\_goal\_.x() = transformed\_plan.back().pose.position.x;

robot\_goal\_.y() = transformed\_plan.back().pose.position.y;

// Overwrite goal orientation if needed// 覆盖目标点的朝向，如果有必要的话

if (cfg\_.trajectory.global\_plan\_overwrite\_orientation)

{

robot\_goal\_.theta() = estimateLocalGoalOrientation(global\_plan\_, transformed\_plan.back(), goal\_idx, tf\_plan\_to\_global);

// overwrite/update goal orientation of the transformed plan with the actual goal (enable using the plan as initialization)

tf2::Quaternion q;

q.setRPY(0, 0, robot\_goal\_.theta());

tf2::convert(q, transformed\_plan.back().pose.orientation);

}

else

{

//直接使用全局路径的朝向

robot\_goal\_.theta() = tf2::getYaw(transformed\_plan.back().pose.orientation);

}

// overwrite/update start of the transformed plan with the actual robot position (allows using the plan as initial trajectory)

if (transformed\_plan.size()==1) // plan only contains the goal// 路径中只有目标点

{

transformed\_plan.insert(transformed\_plan.begin(), geometry\_msgs::PoseStamped()); // insert start (not yet initialized)// 插入起始位姿（还没有初始化）

}

transformed\_plan.front() = robot\_pose; // update start// 更新起始点

// clear currently existing obstacles

obstacles\_.clear();

// Update obstacle container with costmap information or polygons provided by a costmap\_converter plugin

// 用代价地图信息或者costmap\_converter提供的多边形信息来更新障碍物容器

if (costmap\_converter\_)

updateObstacleContainerWithCostmapConverter();

else

updateObstacleContainerWithCostmap();

// also consider custom obstacles (must be called after other updates, since the container is not cleared)

updateObstacleContainerWithCustomObstacles();

// Do not allow config changes during the following optimization step

//配置线程锁,在执行下面命令的过程中，不允许配置被修改

boost::mutex::scoped\_lock cfg\_lock(cfg\_.configMutex());

// Now perform the actual planning// 准备工作做了这么久，现在开始真正的局部轨迹规划 ╮(╯▽╰)╭

// bool success = planner\_->plan(robot\_pose\_, robot\_goal\_, robot\_vel\_, cfg\_.goal\_tolerance.free\_goal\_vel); // straight line init

bool success = planner\_->plan(transformed\_plan, &robot\_vel\_, cfg\_.goal\_tolerance.free\_goal\_vel);

if (!success)

{

planner\_->clearPlanner(); // force reinitialization for next time

ROS\_WARN("teb\_local\_planner was not able to obtain a local plan for the current setting.");

++no\_infeasible\_plans\_; // increase number of infeasible solutions in a row

time\_last\_infeasible\_plan\_ = ros::Time::now();

last\_cmd\_ = cmd\_vel.twist;

message = "teb\_local\_planner was not able to obtain a local plan";

return mbf\_msgs::ExePathResult::NO\_VALID\_CMD;

}

// Check feasibility (but within the first few states only)

if(cfg\_.robot.is\_footprint\_dynamic)

{

// Update footprint of the robot and minimum and maximum distance from the center of the robot to its footprint vertices.

footprint\_spec\_ = costmap\_ros\_->getRobotFootprint();

costmap\_2d::calculateMinAndMaxDistances(footprint\_spec\_, robot\_inscribed\_radius\_, robot\_circumscribed\_radius);

}

bool feasible = planner\_->isTrajectoryFeasible(costmap\_model\_.get(), footprint\_spec\_, robot\_inscribed\_radius\_, robot\_circumscribed\_radius, cfg\_.trajectory.feasibility\_check\_no\_poses);

if (!feasible)

{

cmd\_vel.twist.linear.x = cmd\_vel.twist.linear.y = cmd\_vel.twist.angular.z = 0;

// now we reset everything to start again with the initialization of new trajectories.

planner\_->clearPlanner();

ROS\_WARN("TebLocalPlannerROS: trajectory is not feasible. Resetting planner...");

++no\_infeasible\_plans\_; // increase number of infeasible solutions in a row

time\_last\_infeasible\_plan\_ = ros::Time::now();

last\_cmd\_ = cmd\_vel.twist;

message = "teb\_local\_planner trajectory is not feasible";

return mbf\_msgs::ExePathResult::NO\_VALID\_CMD;

}

// Get the velocity command for this sampling interval

//teb的速度比较简单，就是受cfg\_.trajectory.control\_look\_ahead\_poses影响，取向前多少个位姿点的姿态。然后用当前点的位姿与目标点做差

//速度的话就是两点间距离/前面计算过的点与点之间理想时间和

//在前面运动优化部分有计算过每个位姿之间的理论运动时间。例如这里如果要取前向第五个点，那么时间就是前面计算过的点1-2-3-4-5的时间和，距离就是点1与点5的距离差

//角度的话也是简单明了的两点之间角度差/时间

//https://blog.csdn.net/YiYeZhiNian/article/details/130189348

if (!planner\_->getVelocityCommand(cmd\_vel.twist.linear.x, cmd\_vel.twist.linear.y, cmd\_vel.twist.angular.z, cfg\_.trajectory.control\_look\_ahead\_poses))

{

planner\_->clearPlanner();

ROS\_WARN("TebLocalPlannerROS: velocity command invalid. Resetting planner...");

++no\_infeasible\_plans\_; // increase number of infeasible solutions in a row

time\_last\_infeasible\_plan\_ = ros::Time::now();

last\_cmd\_ = cmd\_vel.twist;

message = "teb\_local\_planner velocity command invalid";

return mbf\_msgs::ExePathResult::NO\_VALID\_CMD;

}

// Saturate velocity, if the optimization results violates the constraints (could be possible due to soft constraints).

//进行约束

saturateVelocity(cmd\_vel.twist.linear.x, cmd\_vel.twist.linear.y, cmd\_vel.twist.angular.z,

cfg\_.robot.max\_vel\_x, cfg\_.robot.max\_vel\_y, cfg\_.robot.max\_vel\_theta, cfg\_.robot.max\_vel\_x\_backwards);

// convert rot-vel to steering angle if desired (carlike robot).

// The min\_turning\_radius is allowed to be slighly smaller since it is a soft-constraint

// and opposed to the other constraints not affected by penalty\_epsilon. The user might add a safety margin to the parameter itself.

if (cfg\_.robot.cmd\_angle\_instead\_rotvel)

{

cmd\_vel.twist.angular.z = convertTransRotVelToSteeringAngle(cmd\_vel.twist.linear.x, cmd\_vel.twist.angular.z,

cfg\_.robot.wheelbase, 0.95\*cfg\_.robot.min\_turning\_radius);

if (!std::isfinite(cmd\_vel.twist.angular.z))

{

cmd\_vel.twist.linear.x = cmd\_vel.twist.linear.y = cmd\_vel.twist.angular.z = 0;

last\_cmd\_ = cmd\_vel.twist;

planner\_->clearPlanner();

ROS\_WARN("TebLocalPlannerROS: Resulting steering angle is not finite. Resetting planner...");

++no\_infeasible\_plans\_; // increase number of infeasible solutions in a row

time\_last\_infeasible\_plan\_ = ros::Time::now();

message = "teb\_local\_planner steering angle is not finite";

return mbf\_msgs::ExePathResult::NO\_VALID\_CMD;

}

}

// a feasible solution should be found, reset counter

no\_infeasible\_plans\_ = 0;

// store last command (for recovery analysis etc.)

last\_cmd\_ = cmd\_vel.twist;

// Now visualize everything//发布可视化话题

planner\_->visualize();

visualization\_->publishObstacles(obstacles\_);

visualization\_->publishViaPoints(via\_points\_);

visualization\_->publishGlobalPlan(global\_plan\_);

return mbf\_msgs::ExePathResult::SUCCESS;

}

这里面主要的功能函数有：

planner\_->plan(transformed\_plan,&robot\_vel\_,cfg\_.goal\_tolerance.free\_goal\_vel);用真正的局部轨迹规划，优化位姿点 和

planner\_->getVelocityCommand(cmd\_vel.twist.linear.x, cmd\_vel.twist.linear.y, cmd\_vel.twist.angular.z, cfg\_.trajectory.control\_look\_ahead\_poses)；用于根据局部规矩优化的位姿点计算控制指令。

* + 1. planner\_所指向子类对象分析

这个类的成员主要包括：



这三个，是初始化时外面指针传入的。



创建teb\_类，主要用于存放待优化和优化的位姿点



创建optimizer\_智能指针，在初始化中会在堆区实例化一个图优化求解器对象，并指向

* + 1. planner\_->plan()分析

源码如下：

bool TebOptimalPlanner::plan(const std::vector<geometry\_msgs::PoseStamped>& initial\_plan, const geometry\_msgs::Twist\* start\_vel, bool free\_goal\_vel)

{

ROS\_ASSERT\_MSG(initialized\_, "Call initialize() first.");

if (!teb\_.isInit())

{

teb\_.initTrajectoryToGoal(initial\_plan, cfg\_->robot.max\_vel\_x, cfg\_->robot.max\_vel\_theta, cfg\_->trajectory.global\_plan\_overwrite\_orientation,

cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion);

}

else // warm start

{

PoseSE2 start\_(initial\_plan.front().pose);//获取起始点位状

PoseSE2 goal\_(initial\_plan.back().pose);//获取终点位状

if (teb\_.sizePoses()>0

&& (goal\_.position() - teb\_.BackPose().position()).norm() < cfg\_->trajectory.force\_reinit\_new\_goal\_dist

&& fabs(g2o::normalize\_theta(goal\_.theta() - teb\_.BackPose().theta())) < cfg\_->trajectory.force\_reinit\_new\_goal\_angular) // actual warm start!

//裁剪已经有的teb路径 裁剪掉node　顶点 更新起点或者终点

teb\_.updateAndPruneTEB(start\_, goal\_, cfg\_->trajectory.min\_samples); // update TEB

else // goal too far away -> reinit

{

ROS\_DEBUG("New goal: distance to existing goal is higher than the specified threshold. Reinitalizing trajectories.");

teb\_.clearTimedElasticBand();

teb\_.initTrajectoryToGoal(initial\_plan, cfg\_->robot.max\_vel\_x, cfg\_->robot.max\_vel\_theta, cfg\_->trajectory.global\_plan\_overwrite\_orientation,

cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion);

}

}

//\*\*\*\*\*\*设定路径初始化速度 setVelocityStart \*\*\*\*\*\*

if (start\_vel)

setVelocityStart(\*start\_vel);

if (free\_goal\_vel)

setVelocityGoalFree();

else

vel\_goal\_.first = true; // we just reactivate and use the previously set velocity (should be zero if nothing was modified)

// now optimize

return optimizeTEB(cfg\_->optim.no\_inner\_iterations, cfg\_->optim.no\_outer\_iterations);

}

bool TebOptimalPlanner::plan(const tf::Pose& start, const tf::Pose& goal, const geometry\_msgs::Twist\* start\_vel, bool free\_goal\_vel)

{

PoseSE2 start\_(start);

PoseSE2 goal\_(goal);

return plan(start\_, goal\_, start\_vel);

}

bool TebOptimalPlanner::plan(const PoseSE2& start, const PoseSE2& goal, const geometry\_msgs::Twist\* start\_vel, bool free\_goal\_vel)

{

//参考我csdn收藏

ROS\_ASSERT\_MSG(initialized\_, "Call initialize() first.");

if (!teb\_.isInit())

{

// init trajectory

teb\_.initTrajectoryToGoal(start, goal, 0, cfg\_->robot.max\_vel\_x, cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion); // 0 intermediate samples, but dt=1 -> autoResize will add more samples before calling first optimization

//初始化函数 initTrajectoryToGoal功能是生成从起点到终点的连线，并在连线上进行采样，填充teb中的顶点。如果已经初始化过了，则判断teb\_中顶点数量是否为0，如果不为0且终点没有发生巨大变化，则使用updateAndPruneTEB函数添加顶点，反之则重新初始化顶点。

}

else // warm start

{

if (teb\_.sizePoses() > 0

&& (goal.position() - teb\_.BackPose().position()).norm() < cfg\_->trajectory.force\_reinit\_new\_goal\_dist

&& fabs(g2o::normalize\_theta(goal.theta() - teb\_.BackPose().theta())) < cfg\_->trajectory.force\_reinit\_new\_goal\_angular) // actual warm start!

teb\_.updateAndPruneTEB(start, goal, cfg\_->trajectory.min\_samples);

else // goal too far away -> reinit

{

ROS\_DEBUG("New goal: distance to existing goal is higher than the specified threshold. Reinitalizing trajectories.");

teb\_.clearTimedElasticBand();

teb\_.initTrajectoryToGoal(start, goal, 0, cfg\_->robot.max\_vel\_x, cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion);

}

}

if (start\_vel)

setVelocityStart(\*start\_vel);

if (free\_goal\_vel)

setVelocityGoalFree();

else

vel\_goal\_.first = true; // we just reactivate and use the previously set velocity (should be zero if nothing was modified)

// now optimize

return optimizeTEB(cfg\_->optim.no\_inner\_iterations, cfg\_->optim.no\_outer\_iterations);//调用最核心的凸优化代码进行局部规划

}

内部核心成员函数：optimizeTEB(cfg\_->optim.no\_inner\_iterations, cfg\_->optim.no\_outer\_iterations);

如下：包括外循环和内循环

单次外循环包括：

5.3.3.1 success = buildGraph(weight\_multiplier);//建图过程函数

添加位姿点作为顶点：

void TebOptimalPlanner::AddTEBVertices()

{

// add vertices to graph

ROS\_DEBUG\_COND(cfg\_->optim.optimization\_verbose, "Adding TEB vertices ...");

unsigned int id\_counter = 0; // used for vertices ids

obstacles\_per\_vertex\_.resize(teb\_.sizePoses());//设置obstacles\_per\_vertex\_大小

auto iter\_obstacle = obstacles\_per\_vertex\_.begin();

for (int i=0; i<teb\_.sizePoses(); ++i)

{

teb\_.PoseVertex(i)->setId(id\_counter++);

optimizer\_->addVertex(teb\_.PoseVertex(i));//将teb\_中位置顶点加入到凸优化定点序列

if (teb\_.sizeTimeDiffs()!=0 && i<teb\_.sizeTimeDiffs())

{

teb\_.TimeDiffVertex(i)->setId(id\_counter++);

optimizer\_->addVertex(teb\_.TimeDiffVertex(i));

}

iter\_obstacle->clear();

(iter\_obstacle++)->reserve(obstacles\_->size());

}

}

添加障碍物边：

void TebOptimalPlanner::AddEdgesObstacles(double weight\_multiplier)

{

if (cfg\_->optim.weight\_obstacle==0 || weight\_multiplier==0 || obstacles\_==nullptr )

return; // if weight equals zero skip adding edges!

bool inflated = cfg\_->obstacles.inflation\_dist > cfg\_->obstacles.min\_obstacle\_dist;

Eigen::Matrix<double,1,1> information;

information.fill(cfg\_->optim.weight\_obstacle \* weight\_multiplier);

Eigen::Matrix<double,2,2> information\_inflated;

information\_inflated(0,0) = cfg\_->optim.weight\_obstacle \* weight\_multiplier;

information\_inflated(1,1) = cfg\_->optim.weight\_inflation;

information\_inflated(0,1) = information\_inflated(1,0) = 0;

auto iter\_obstacle = obstacles\_per\_vertex\_.begin();

//定义了一个lambda函数，后面使用

auto create\_edge = [inflated, &information, &information\_inflated, this] (int index, const Obstacle\* obstacle) {

if (inflated)

{

EdgeInflatedObstacle\* dist\_bandpt\_obst = new EdgeInflatedObstacle;

dist\_bandpt\_obst->setVertex(0,teb\_.PoseVertex(index));//设置障碍物关联的顶点

dist\_bandpt\_obst->setInformation(information\_inflated);

dist\_bandpt\_obst->setParameters(\*cfg\_, robot\_model\_.get(), obstacle);//设置约束参数

optimizer\_->addEdge(dist\_bandpt\_obst);

}

else

{

EdgeObstacle\* dist\_bandpt\_obst = new EdgeObstacle;

dist\_bandpt\_obst->setVertex(0,teb\_.PoseVertex(index));

dist\_bandpt\_obst->setInformation(information);

dist\_bandpt\_obst->setParameters(\*cfg\_, robot\_model\_.get(), obstacle);

optimizer\_->addEdge(dist\_bandpt\_obst);

};

};

// iterate all teb points, skipping the last and, if the EdgeVelocityObstacleRatio edges should not be created, the first one too

const int first\_vertex = cfg\_->optim.weight\_velocity\_obstacle\_ratio == 0 ? 1 : 0;

for (int i = first\_vertex; i < teb\_.sizePoses() - 1; ++i)//遍历每个位姿点

{

double left\_min\_dist = std::numeric\_limits<double>::max();

double right\_min\_dist = std::numeric\_limits<double>::max();

ObstaclePtr left\_obstacle;

ObstaclePtr right\_obstacle;

const Eigen::Vector2d pose\_orient = teb\_.Pose(i).orientationUnitVec();//获取第i个点的位置坐标

// iterate obstacles

for (const ObstaclePtr& obst : \*obstacles\_)

{

// we handle dynamic obstacles differently below

if(cfg\_->obstacles.include\_dynamic\_obstacles && obst->isDynamic())

continue;

// calculate distance to robot model

double dist = robot\_model\_->calculateDistance(teb\_.Pose(i), obst.get());//计算障碍物和位姿点坐标距离

// force considering obstacle if really close to the current pose

if (dist < cfg\_->obstacles.min\_obstacle\_dist\*cfg\_->obstacles.obstacle\_association\_force\_inclusion\_factor)//如果距离小于设定的安全距离，认为该障碍物有危险，加入到iter\_obstacle数组

{

iter\_obstacle->push\_back(obst);

continue;

}

// cut-off distance

if (dist > cfg\_->obstacles.min\_obstacle\_dist\*cfg\_->obstacles.obstacle\_association\_cutoff\_factor)

continue;

// determine side (left or right) and assign obstacle if closer than the previous one

if (cross2d(pose\_orient, obst->getCentroid()) > 0) // left

{

if (dist < left\_min\_dist)

{

left\_min\_dist = dist;

left\_obstacle = obst;

}

}

else

{

if (dist < right\_min\_dist)

{

right\_min\_dist = dist;

right\_obstacle = obst;

}

}

}

if (left\_obstacle)

iter\_obstacle->push\_back(left\_obstacle);

if (right\_obstacle)

iter\_obstacle->push\_back(right\_obstacle);

// continue here to ignore obstacles for the first pose, but use them later to create the EdgeVelocityObstacleRatio edges

if (i == 0)

{

++iter\_obstacle;

continue;

}

// create obstacle edges

for (const ObstaclePtr obst : \*iter\_obstacle)//iter\_obstacle里面都是第i个位姿点的障碍物，将iter\_obstacle数组进行遍历，把他们加入到障碍物边

create\_edge(i, obst.get());

++iter\_obstacle;

}

}

添加路径点边：接下来是添加经过路径点(必定要经过的)边，目的是减小坐标点离路径点的距离

源码如下：

void TebOptimalPlanner::AddEdgesViaPoints()

{

if (cfg\_->optim.weight\_viapoint==0 || via\_points\_==NULL || via\_points\_->empty() )

return; // if weight equals zero skip adding edges!

int start\_pose\_idx = 0;

int n = teb\_.sizePoses();

if (n<3) // we do not have any degrees of freedom for reaching via-points

return;

for (ViaPointContainer::const\_iterator vp\_it = via\_points\_->begin(); vp\_it != via\_points\_->end(); ++vp\_it)//遍历需要经过的点

{

int index = teb\_.findClosestTrajectoryPose(\*vp\_it, NULL, start\_pose\_idx);//寻找最近轨迹点的下标

if (cfg\_->trajectory.via\_points\_ordered)

start\_pose\_idx = index+2; // skip a point to have a DOF inbetween for further via-points

// check if point conicides with goal or is located behind it

if ( index > n-2 )

index = n-2; // set to a pose before the goal, since we can move it away!

// check if point coincides with start or is located before it

if ( index < 1)

{

if (cfg\_->trajectory.via\_points\_ordered)

{

index = 1; // try to connect the via point with the second (and non-fixed) pose. It is likely that autoresize adds new poses inbetween later.

}

else

{

ROS\_DEBUG("TebOptimalPlanner::AddEdgesViaPoints(): skipping a via-point that is close or behind the current robot pose.");

continue; // skip via points really close or behind the current robot pose

}

}

Eigen::Matrix<double,1,1> information;

information.fill(cfg\_->optim.weight\_viapoint);//配置权重

EdgeViaPoint\* edge\_viapoint = new EdgeViaPoint;

edge\_viapoint->setVertex(0,teb\_.PoseVertex(index));//设置该经过关联的顶点

edge\_viapoint->setInformation(information);

edge\_viapoint->setParameters(\*cfg\_, &(\*vp\_it));//\*vp\_it传入

optimizer\_->addEdge(edge\_viapoint);

}

}

添加速度边，速度包括了线速度和角速度两个维度，目的是防止线速度和角速度超过给定阈值：

void TebOptimalPlanner::AddEdgesVelocity()

{

if (cfg\_->robot.max\_vel\_y == 0) // non-holonomic robot

{

if ( cfg\_->optim.weight\_max\_vel\_x==0 && cfg\_->optim.weight\_max\_vel\_theta==0)

return; // if weight equals zero skip adding edges!

int n = teb\_.sizePoses();//待优化位姿个数

Eigen::Matrix<double,2,2> information;

information(0,0) = cfg\_->optim.weight\_max\_vel\_x;

information(1,1) = cfg\_->optim.weight\_max\_vel\_theta;

information(0,1) = 0.0;

information(1,0) = 0.0;

for (int i=0; i < n - 1; ++i)

{

EdgeVelocity\* velocity\_edge = new EdgeVelocity;

velocity\_edge->setVertex(0,teb\_.PoseVertex(i));

velocity\_edge->setVertex(1,teb\_.PoseVertex(i+1));

velocity\_edge->setVertex(2,teb\_.TimeDiffVertex(i));

velocity\_edge->setInformation(information);

velocity\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(velocity\_edge);

}

}

else // holonomic-robot//我们选择这个

{

if ( cfg\_->optim.weight\_max\_vel\_x==0 && cfg\_->optim.weight\_max\_vel\_y==0 && cfg\_->optim.weight\_max\_vel\_theta==0)

return; // if weight equals zero skip adding edges!

int n = teb\_.sizePoses();

Eigen::Matrix<double,3,3> information;//权重配置矩阵

information.fill(0);

information(0,0) = cfg\_->optim.weight\_max\_vel\_x;

information(1,1) = cfg\_->optim.weight\_max\_vel\_y;

information(2,2) = cfg\_->optim.weight\_max\_vel\_theta;

for (int i=0; i < n - 1; ++i)

{

EdgeVelocityHolonomic\* velocity\_edge = new EdgeVelocityHolonomic;

velocity\_edge->setVertex(0,teb\_.PoseVertex(i));

velocity\_edge->setVertex(1,teb\_.PoseVertex(i+1));

velocity\_edge->setVertex(2,teb\_.TimeDiffVertex(i));//速度边关联有三个顶点，其中两个位姿顶点，一个时间间隔顶点

velocity\_edge->setInformation(information);//设置权重

velocity\_edge->setTebConfig(\*cfg\_);//具体的速度限制应该放在\*cfg\_里面

optimizer\_->addEdge(velocity\_edge);

}

}

}

添加加速度边，同样包含了线加速度和角加速度，目的是防止加速度超过给定阈值：

void TebOptimalPlanner::AddEdgesAcceleration()

{

if (cfg\_->optim.weight\_acc\_lim\_x==0 && cfg\_->optim.weight\_acc\_lim\_theta==0)

return; // if weight equals zero skip adding edges!

int n = teb\_.sizePoses();

if (cfg\_->robot.max\_vel\_y == 0 || cfg\_->robot.acc\_lim\_y == 0) // non-holonomic robot

{

Eigen::Matrix<double,2,2> information;

information.fill(0);

information(0,0) = cfg\_->optim.weight\_acc\_lim\_x;

information(1,1) = cfg\_->optim.weight\_acc\_lim\_theta;

// check if an initial velocity should be taken into accound

if (vel\_start\_.first)

{

EdgeAccelerationStart\* acceleration\_edge = new EdgeAccelerationStart;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(0));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(1));

acceleration\_edge->setVertex(2,teb\_.TimeDiffVertex(0));

acceleration\_edge->setInitialVelocity(vel\_start\_.second);

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

// now add the usual acceleration edge for each tuple of three teb poses

for (int i=0; i < n - 2; ++i)

{

EdgeAcceleration\* acceleration\_edge = new EdgeAcceleration;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(i));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(i+1));

acceleration\_edge->setVertex(2,teb\_.PoseVertex(i+2));

acceleration\_edge->setVertex(3,teb\_.TimeDiffVertex(i));

acceleration\_edge->setVertex(4,teb\_.TimeDiffVertex(i+1));

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

// check if a goal velocity should be taken into accound

if (vel\_goal\_.first)

{

EdgeAccelerationGoal\* acceleration\_edge = new EdgeAccelerationGoal;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(n-2));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(n-1));

acceleration\_edge->setVertex(2,teb\_.TimeDiffVertex( teb\_.sizeTimeDiffs()-1 ));

acceleration\_edge->setGoalVelocity(vel\_goal\_.second);

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

}

else // holonomic robot选择这个

{

Eigen::Matrix<double,3,3> information;//权重设置

information.fill(0);

information(0,0) = cfg\_->optim.weight\_acc\_lim\_x;

information(1,1) = cfg\_->optim.weight\_acc\_lim\_y;

information(2,2) = cfg\_->optim.weight\_acc\_lim\_theta;

// check if an initial velocity should be taken into accound

if (vel\_start\_.first)

{

EdgeAccelerationHolonomicStart\* acceleration\_edge = new EdgeAccelerationHolonomicStart;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(0));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(1));

acceleration\_edge->setVertex(2,teb\_.TimeDiffVertex(0));

acceleration\_edge->setInitialVelocity(vel\_start\_.second);

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

// now add the usual acceleration edge for each tuple of three teb poses

for (int i=0; i < n - 2; ++i)

{

EdgeAccelerationHolonomic\* acceleration\_edge = new EdgeAccelerationHolonomic;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(i));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(i+1));

acceleration\_edge->setVertex(2,teb\_.PoseVertex(i+2));

acceleration\_edge->setVertex(3,teb\_.TimeDiffVertex(i));

acceleration\_edge->setVertex(4,teb\_.TimeDiffVertex(i+1));

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

// check if a goal velocity should be taken into accound

if (vel\_goal\_.first)

{

EdgeAccelerationHolonomicGoal\* acceleration\_edge = new EdgeAccelerationHolonomicGoal;

acceleration\_edge->setVertex(0,teb\_.PoseVertex(n-2));

acceleration\_edge->setVertex(1,teb\_.PoseVertex(n-1));

acceleration\_edge->setVertex(2,teb\_.TimeDiffVertex( teb\_.sizeTimeDiffs()-1 ));

acceleration\_edge->setGoalVelocity(vel\_goal\_.second);

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(acceleration\_edge);

}

}

}

添加时间最优化边，用于优化时间：时间也是待优化的顶点，要求总时间较短：

void TebOptimalPlanner::AddEdgesTimeOptimal()

{

if (cfg\_->optim.weight\_optimaltime==0)

return; // if weight equals zero skip adding edges!

Eigen::Matrix<double,1,1> information;

information.fill(cfg\_->optim.weight\_optimaltime);//权重设置

for (int i=0; i < teb\_.sizeTimeDiffs(); ++i)//遍历每一段时间

{

EdgeTimeOptimal\* timeoptimal\_edge = new EdgeTimeOptimal;

timeoptimal\_edge->setVertex(0,teb\_.TimeDiffVertex(i));//加入时间顶点

timeoptimal\_edge->setInformation(information);//加入权重

timeoptimal\_edge->setTebConfig(\*cfg\_);//设置配置参数

optimizer\_->addEdge(timeoptimal\_edge);

}

}

添加添加轨迹最短化边，目的是优化轨迹长度:

void TebOptimalPlanner::AddEdgesShortestPath()

{

if (cfg\_->optim.weight\_shortest\_path==0)

return; // if weight equals zero skip adding edges!

Eigen::Matrix<double,1,1> information;//权重矩阵

information.fill(cfg\_->optim.weight\_shortest\_path);

for (int i=0; i < teb\_.sizePoses()-1; ++i)//每个位姿点进行遍历

{

EdgeShortestPath\* shortest\_path\_edge = new EdgeShortestPath;

shortest\_path\_edge->setVertex(0,teb\_.PoseVertex(i));

shortest\_path\_edge->setVertex(1,teb\_.PoseVertex(i+1));//每条最短路径边关联了两个顶点

shortest\_path\_edge->setInformation(information);

shortest\_path\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(shortest\_path\_edge);

}

}

添加动力学限制边，目的是让生成的轨迹满足运动学约束:

void TebOptimalPlanner::AddEdgesKinematicsCarlike()

{//AddEdgesKinematicsDiffDrive函数的内容如下所示，对于每两个相邻坐标顶点构建 EdgeKinematicsDiffDrive

//对象， EdgeKinematicsDiffDrive是两顶点边，并且拥有雅克比计算的定义。

//可能根据最大前轮转角变化率和最大前轮转角对位姿点约束

if (cfg\_->optim.weight\_kinematics\_nh==0 && cfg\_->optim.weight\_kinematics\_turning\_radius==0)

return; // if weight equals zero skip adding edges!

// create edge for satisfiying kinematic constraints

Eigen::Matrix<double,2,2> information\_kinematics;

information\_kinematics.fill(0.0);

information\_kinematics(0, 0) = cfg\_->optim.weight\_kinematics\_nh;

information\_kinematics(1, 1) = cfg\_->optim.weight\_kinematics\_turning\_radius;

for (int i=0; i < teb\_.sizePoses()-1; i++) // ignore twiced start only

{

EdgeKinematicsCarlike\* kinematics\_edge = new EdgeKinematicsCarlike;

kinematics\_edge->setVertex(0,teb\_.PoseVertex(i));

kinematics\_edge->setVertex(1,teb\_.PoseVertex(i+1));

kinematics\_edge->setInformation(information\_kinematics);

kinematics\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(kinematics\_edge);

}

}

添加朝向倾向性，用于使机器人在避障过程中倾向左侧还是右侧:

void TebOptimalPlanner::AddEdgesPreferRotDir()

{

//TODO(roesmann): Note, these edges can result in odd predictions, in particular

// we can observe a substantional mismatch between open- and closed-loop planning

// leading to a poor control performance.

// At the moment, we keep these functionality for oscillation recovery:

// Activating the edge for a short time period might not be crucial and

// could move the robot to a new oscillation-free state.

// This needs to be analyzed in more detail!

if (prefer\_rotdir\_ == RotType::none || cfg\_->optim.weight\_prefer\_rotdir==0)

return; // if weight equals zero skip adding edges!

if (prefer\_rotdir\_ != RotType::right && prefer\_rotdir\_ != RotType::left)

{

ROS\_WARN("TebOptimalPlanner::AddEdgesPreferRotDir(): unsupported RotType selected. Skipping edge creation.");

return;

}

// create edge for satisfiying kinematic constraints

Eigen::Matrix<double,1,1> information\_rotdir;

information\_rotdir.fill(cfg\_->optim.weight\_prefer\_rotdir);

for (int i=0; i < teb\_.sizePoses()-1 && i < 3; ++i) // currently: apply to first 3 rotations

{

EdgePreferRotDir\* rotdir\_edge = new EdgePreferRotDir;

rotdir\_edge->setVertex(0,teb\_.PoseVertex(i));

rotdir\_edge->setVertex(1,teb\_.PoseVertex(i+1));

rotdir\_edge->setInformation(information\_rotdir);

if (prefer\_rotdir\_ == RotType::left)

rotdir\_edge->preferLeft();

else if (prefer\_rotdir\_ == RotType::right)

rotdir\_edge->preferRight();

optimizer\_->addEdge(rotdir\_edge);

}

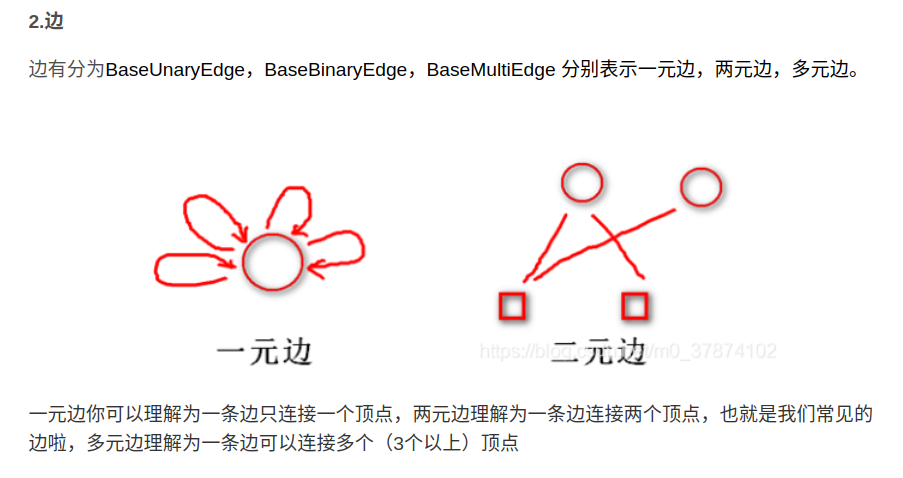
}

利用optimizer\_->addEdge(rotdir\_edge);把边添加图优化器，边需要满足一定的规则，需要继承g2o::BaseBinaryEdge<D, E, VertexXi, VertexXj>，创建BaseTebBinaryEdge类，然后再根据需求创建上面这些EdgeObstacle,EdgeViaPoint,EdgeVelocityHolonomic,EdgeKinematicsCarlike,EdgePreferRotDir等子子类，在BaseTebBinaryEdge类中会根据需求对g2o::BaseBinaryEdge<D, E, VertexXi, VertexXj>中虚函数重写，子子类也会根据需求在BaseTebBinaryEdge类中会根据需求对g2o::BaseBinaryEdge<D, E, VertexXi, VertexXj>中虚函数重写，最终实现多态。

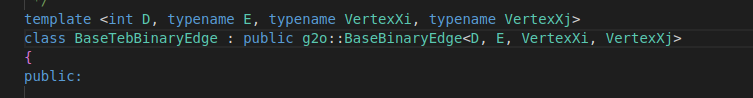
先了解一下图优化库：

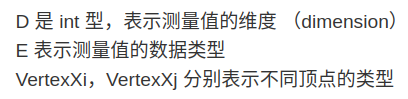
参考：<https://blog.csdn.net/m0_37874102/article/details/114291951>

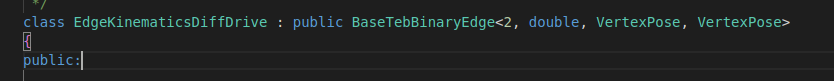
边的类型有：



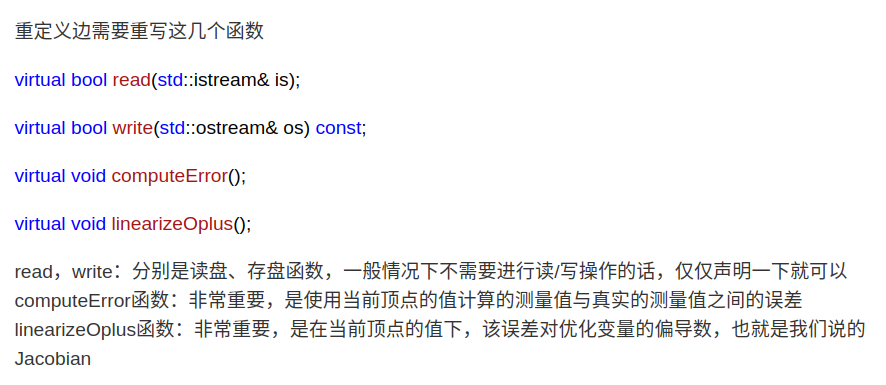
因此图优化基类用的是类模板编程：



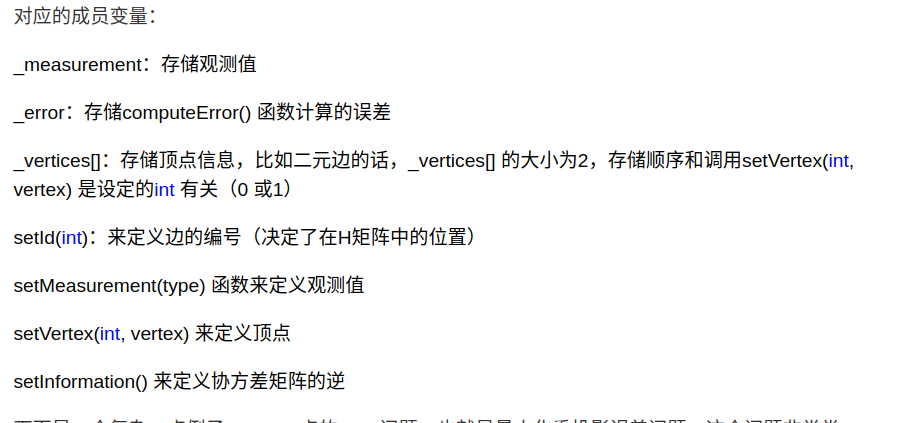
，D为\_error的维度



在子类中需要重写以下虚函数，实现多态

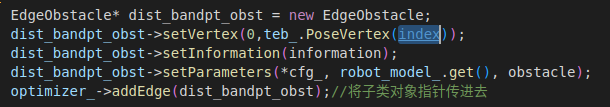


图优化基类中的成员还有：

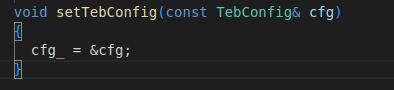


核心来了：这是图优化库的边基类，边子类需要我们自己设计，其目的是通过对基类函数重写，并设计一些其他的成员，来实现对基类成员变量,,进行填充，然后将子类对象的指针传入图优化器对象，如optimizer\_->addEdge(dist\_bandpt\_obst)，dist\_bandpt\_obst是子类对象指针，optimizer\_中有个边父类指针接收，这样optimizer\_就可以利用该父类指针获得上面填充的成员，用于进一步计算。

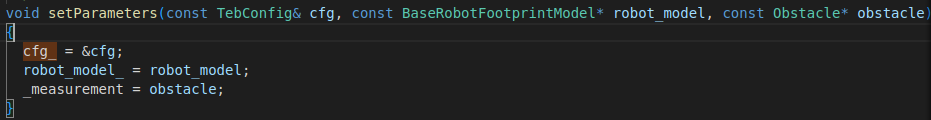
重要举例：添加障碍物边到图优化器



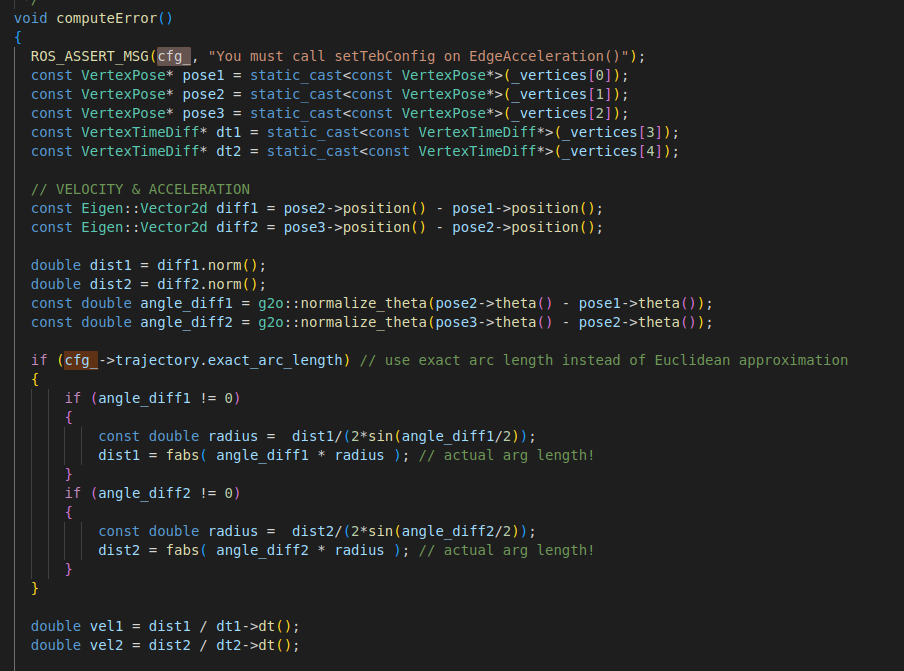
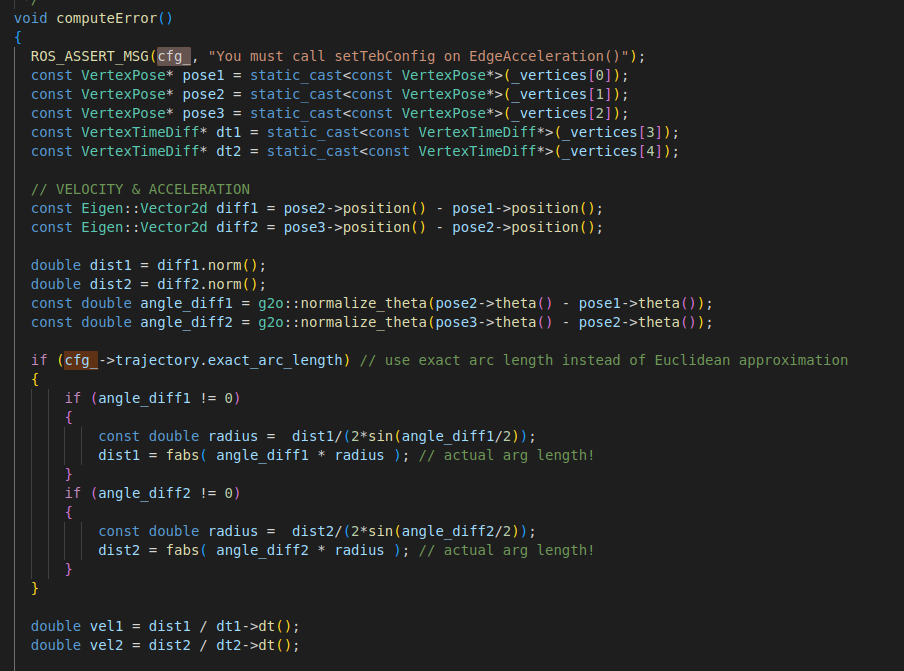
EdgeObstacle是子类，它会继承BaseTebUnaryEdge类，而BaseTebUnaryEdge类会继承g2o::BaseUnaryEdge<D, E, VertexXi>基类。BaseTebUnaryEdge类中的东西是各子类共有的部分，因此还特意创建了个父类，在子类BaseTebUnaryEdge中主要是对基类虚函数computeError()，linearizeOplus()进行重写，来填充，以及在BaseTebUnaryEdge类中定义了一些其他成员函数都是为了便于基类成员的填充，比如setTebConfig(const TebConfig& cfg)



这个在添加加速度约束边的时候有，添加障碍物边的时候是直接在子类设计了以下成员函数添加



但做这些工作都是为了边基类成员变量的添加，cfg\_不是边基类成员，在computeError()函数里面会使用计算



边子类指针给到图优化器后，在真正进行优化的时候利用多态循环各种边的optimizer\_->父类指针->computeError()来计算各自的\_error,后面optimizer\_->父类指针->\_error即可调用各个边的\_error用于进一步计算和优化工作。

5.3.3.12 success = optimizeGraph(iterations\_innerloop, false);//对图进行优化，会进行iterations\_innerloop次内循环

bool TebOptimalPlanner::optimizeGraph(int no\_iterations,bool clear\_after)

{

if (cfg\_->robot.max\_vel\_x<0.01)

{

ROS\_WARN("optimizeGraph(): Robot Max Velocity is smaller than 0.01m/s. Optimizing aborted...");

if (clear\_after) clearGraph();

return false;

}

if (!teb\_.isInit() || teb\_.sizePoses() < cfg\_->trajectory.min\_samples)

{

ROS\_WARN("optimizeGraph(): TEB is empty or has too less elements. Skipping optimization.");

if (clear\_after) clearGraph();

return false;

}

optimizer\_->setVerbose(cfg\_->optim.optimization\_verbose);

optimizer\_->initializeOptimization();

int iter = optimizer\_->optimize(no\_iterations);

// Save Hessian for visualization

// g2o::OptimizationAlgorithmLevenberg\* lm = dynamic\_cast<g2o::OptimizationAlgorithmLevenberg\*> (optimizer\_->solver());

// lm->solver()->saveHessian("~/MasterThesis/Matlab/Hessian.txt");

if(!iter)

{

ROS\_ERROR("optimizeGraph(): Optimization failed! iter=%i", iter);

return false;

}

if (clear\_after) clearGraph();

return true;

}

iter = optimizer\_->optimize(no\_iterations);启动优化器，no\_iterations是内循环迭代次数，我猜是梯度下降，LM或者高斯牛顿法计算的迭代次数限制。

优化后的位姿点会放回到teb\_的pose\_vec\_内指针所指向的内存。

* + 1. planner\_->getVelocityCommand()分析,求解vx，vy，omega，引用传递

源码如下：

bool TebOptimalPlanner::getVelocityCommand(double& vx, double& vy, double& omega, int look\_ahead\_poses) const

{

if (teb\_.sizePoses()<2)

{

ROS\_ERROR("TebOptimalPlanner::getVelocityCommand(): The trajectory contains less than 2 poses. Make sure to init and optimize/plan the trajectory fist.");

vx = 0;

vy = 0;

omega = 0;

return false;

}

look\_ahead\_poses = std::max(1, std::min(look\_ahead\_poses, teb\_.sizePoses() - 1));//对look\_ahead\_poses做一些约束，std::min(look\_ahead\_poses, teb\_.sizePoses() - 1)是为了放在终点附近没有点了

double dt = 0.0;

for(int counter = 0; counter < look\_ahead\_poses; ++counter)

{

dt += teb\_.TimeDiff(counter);//look\_ahead\_poses个位姿点间dt累加

if(dt >= cfg\_->trajectory.dt\_ref \* look\_ahead\_poses) // TODO: change to look-ahead time? Refine trajectory?

{

look\_ahead\_poses = counter + 1;

break;

}

if (dt<=0)

{

ROS\_ERROR("TebOptimalPlanner::getVelocityCommand() - timediff<=0 is invalid!");

vx = 0;

vy = 0;

omega = 0;

return false;

}

// Get velocity from the first two configurations

extractVelocity(teb\_.Pose(0), teb\_.Pose(look\_ahead\_poses), dt, vx, vy, omega);

//图优化结果会放在pose\_vec\_

//teb\_.Pose(0)为当前位姿，teb\_.Pose(look\_ahead\_poses)前面第look\_ahead\_poses个点的位姿，dt为两个位姿点间理论时间，vx, vy, omega为需要求解的

return true;

}

利用小车运动学进行 vx，vy，omega的结算解算：

void TebOptimalPlanner::extractVelocity(const PoseSE2& pose1, const PoseSE2& pose2, double dt, double& vx, double& vy, double& omega) const

{

if (dt == 0)

{

vx = 0;

vy = 0;

omega = 0;

return;

}

Eigen::Vector2d deltaS = pose2.position() - pose1.position();//是一个二维向量

if (cfg\_->robot.max\_vel\_y == 0) // nonholonomic robot//差速轮，没有y方向的速度

{

Eigen::Vector2d conf1dir( cos(pose1.theta()), sin(pose1.theta()) );

// translational velocity

double dir = deltaS.dot(conf1dir);

vx = (double) g2o::sign(dir) \* deltaS.norm()/dt;

vy = 0;

}

else // holonomic robot

{

// transform pose 2 into the current robot frame (pose1)

// for velocities only the rotation of the direction vector is necessary.

// (map->pose1-frame: inverse 2d rotation matrix)

double cos\_theta1 = std::cos(pose1.theta());

double sin\_theta1 = std::sin(pose1.theta());

double p1\_dx = cos\_theta1\*deltaS.x() + sin\_theta1\*deltaS.y();//相当于向量在小车坐标系x轴上投影

double p1\_dy = -sin\_theta1\*deltaS.x() + cos\_theta1\*deltaS.y();//相当于向量在小车坐标系y轴上投影

vx = p1\_dx / dt;//小车坐标系下，小车x方向速度

vy = p1\_dy / dt;//小车坐标系下，小车y方向速度

}

// rotational velocity

//求旋转的角速度

double orientdiff = g2o::normalize\_theta(pose2.theta() - pose1.theta());

omega = orientdiff/dt;

}

由于是引用传入，填充完cmd\_vel后，然后对他们进行约束即可，不需进行发布话题，发布话题也只是为了方便visualization。

